CHAPTER 17

Configuring a Web Server

IN THIS CHAPTER

- Installing an Apache web server
- Configuring Apache
- Securing Apache with iptables and SELinux
- Creating virtual hosts
- Building a secure (HTTPS) website
- Checking Apache for errors

Web servers are responsible for serving up the content you view on the Internet every day. By far, the most popular web server is the Apache (HTTPD) web server, which is sponsored by the Apache Software Foundation (`http://apache.org`). Because Apache is an open source project, it is available with every major Linux distribution, including Fedora, RHEL, and Ubuntu.

You can configure a basic web server to run in Linux in just a few minutes. However, you can configure your Apache web server in a tremendous number of ways. You can configure an Apache web server to serve content for multiple domains (virtual hosting), provide encrypted communications (HTTPS), and secure some or all of a website using different kinds of authentication.

This chapter takes you through the steps to install and configure an Apache web server. These steps include procedures for securing your server as well as using a variety of modules so that you can incorporate different authentication methods and scripting languages into your web server. Then I describe how to generate certificates to create an HTTPS Secure Sockets Layer (SSL) website.

Understanding the Apache Web Server

Apache HTTPD (also known as the *Apache HTTPD Server*) provides the service with which the client web browsers communicate. The daemon process (`httpd`) runs in the background on your server and waits for requests from web clients. Web browsers provide those connections to the HTTP daemon and send requests, which the daemon interprets, sending back the appropriate data (such as a web page or other content).

Linux® Bible, Tenth Edition. Christopher Negus.

Apache HTTPD includes an interface that allows modules to tie into the process to handle specific portions of a request. Among other things, modules are available to handle the processing of scripting languages, such as Perl or PHP, within web documents and to add encryption to connections between clients and the server.

Apache began as a collection of patches and improvements from the National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana-Champaign, to the HTTP daemon. The NCSA HTTP daemon was the most popular HTTP server at the time, but it had started to show its age after its author, Rob McCool, left NCSA in mid-1994.

NOTE
Another project that came from NCSA is Mosaic. Most modern web browsers can trace their origins to Mosaic.

In early 1995, a group of developers formed the Apache Group and began making extensive modifications to the NCSA HTTPD code base. Apache soon replaced NCSA HTTPD as the most popular web server, a title it still holds today.

The Apache Group later formed the Apache Software Foundation (ASF) to promote the development of Apache and other free software. With the start of new projects at ASF, the Apache server became known as Apache HTTPD, although the two terms are still used interchangeably. Currently, ASF has more than 350 open source initiatives, including Tomcat (which includes open source Java Servlet and JavaServer Pages technologies), Hadoop (a project providing highly available, distributed computing), and SpamAssassin (an email filtering program).

Getting and Installing Your Web Server

Although Apache is available with every major Linux distribution, it is often packaged in different ways. In most cases, all you need to start a simple Apache web server is the package containing the Apache daemon itself (`/usr/sbin/httpd`) and its related files. In Fedora, RHEL, and others, the Apache web server comes in the `httpd` package.

Understanding the httpd package

To examine the `httpd` package in Fedora or RHEL before you install it, download the package using the `yumdownloader` command and run a few `rpm` commands on it to view its contents:

```
# yumdownloader httpd
# rpm -qpi httpd-*rpm
Name        : httpd
Version     : 2.4.41
Release     : 1.fc30
Architecture: x86_64
Install Date: (not installed)
```

```
Group       : Unspecified
Size        : 5070831
License     : ASL 2.0
Signature   : RSA/SHA256, Mon 19 Aug 2019 06:06:09 AM EDT, Key ID
ef3c111fcfc659b9
Source RPM  : httpd-2.4.41-1.fc30.src.rpm
Build Date  : Thu 15 Aug 2019 06:07:29 PM EDT
Build Host  : buildvm-30.phx2.fedoraproject.org
Relocations : (not relocatable)
Packager    : Fedora Project
Vendor      : Fedora Project
URL         : http://httpd.apache.org/
Bug URL     : https://bugz.fedoraproject.org/httpd
Summary     : Apache HTTP Server
Description :
The Apache HTTP Server is a powerful, efficient, and extensible
web server.
```

The `yumdownloader` command downloads the latest version of the `httpd` package to the current directory. The `rpm -qpi` command queries the `httpd` RPM package you just downloaded for information. You can see that the package was created by the Fedora project and that it is indeed the Apache HTTP Server package. Next, look inside the package to see the configuration files:

```
# rpm -qpc httpd-*rpm
/etc/httpd/conf.d/autoindex.conf
/etc/httpd/conf.d/userdir.conf
/etc/httpd/conf.d/welcome.conf
/etc/httpd/conf.modules.d/00-base.conf
/etc/httpd/conf.modules.d/00-dav.conf
...
/etc/httpd/conf/httpd.conf
/etc/httpd/conf/magic
/etc/logrotate.d/httpd
/etc/sysconfig/htcacheclean
```

The main configuration file is `/etc/httpd/conf/httpd.conf` for Apache. The `welcome.conf` file defines the default home page for your website, until you add some content. The `magic` file defines rules that the server can use to figure out a file's type when the server tries to open it.

The `/etc/logrotate.d/httpd` file defines how log files produced by Apache are rotated. The `/usr/lib/tmpfiles.d/httpd.conf` file defines a directory that contains temporary runtime files (no need to change that file).

Some Apache modules drop configuration files (`*.conf`) into the `/etc/httpd/conf.modules.d/` directory. Any file in that directory that ends in `.conf` is pulled into the main `httpd.conf` file and used to configure Apache. Most module packages that come with configuration files put those configuration files in the `/etc/httpd/conf.d` directory. For

example, the `mod_ssl` (for secure web servers) and `mod_python` (for interpreting python code) modules have related configuration files in the `/etc/httpd/conf.d` directory named `ssl.conf` and `python.conf`, respectively.

You can just install the `httpd` package to begin setting up your web server. However, you might prefer to add some other packages that are often associated with the `httpd` package. One way to do that is to install the entire Web Server (in Fedora) or Basic Web Server group (in RHEL), as in the following example:

```
# yum groupinstall "Web Server"
```

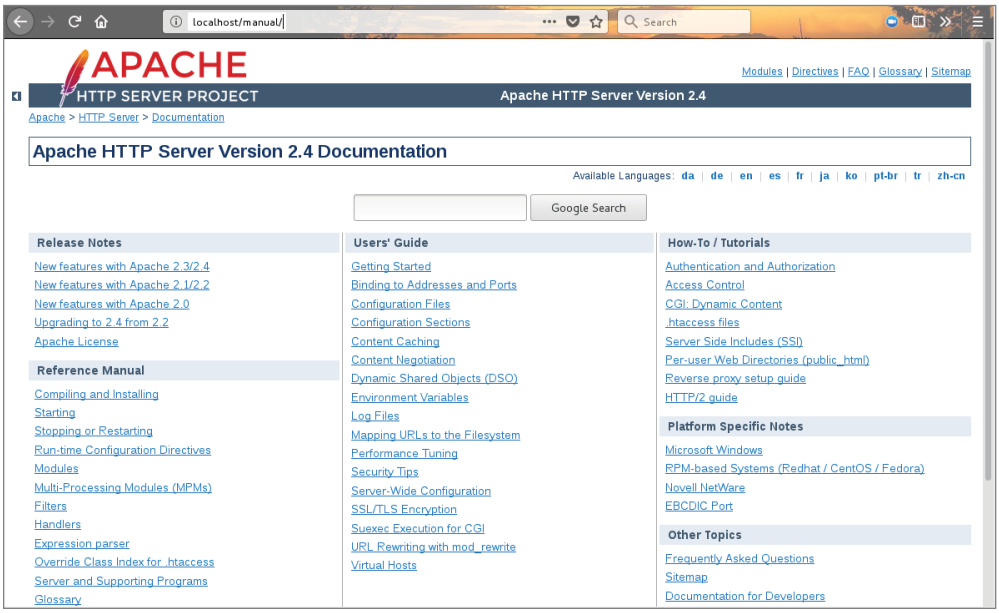
Besides installing some packages that are peripheral to `httpd` (such as `rsyslogd`, `irqbalance`, and others), here are other packages in the Web Server group in Fedora that you get by default along with `httpd`:

`httpd-manual` Fills the `/var/www/manual` directory with the Apache documentation manuals. After you start the httpd service (as shown in later steps), you can access this set of manuals from a web browser on the local machine by typing **`http://localhost/manual`** into the location box.

Externally, instead of localhost, you could use the fully qualified domain name or IP address of the system. The Apache Documentation screen then appears, as shown in Figure 17.1.

FIGURE 17.1

Access Apache documentation directly from the local Apache server.



mod_ssl Contains the module and configuration file needed for the web server to provide secure connections to clients using Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. These features are necessary if you need encrypted communications for online shopping or other data that you want to keep private. The configuration file is located at `/etc/httpd/conf.d/ssl.conf`.

crypto-utils Contains commands for generating keys and certificates needed to do secure communications with the Apache web server.

mod_perl Contains the Perl module (`mod_perl`), configuration file, and associated files needed to allow the Apache web server to execute any Perl code directly.

php Contains the PHP module and configuration file needed to run PHP scripts directly in Apache. Related packages include `php-ldap` (for running PHP code that needs to access LDAP databases) and `php-mysql` (to add database support to the Apache server).

php-ldap Adds support for Lightweight Directory Access Protocol (LDAP) to the PHP module, allowing directory service access over networks.

squid Provides proxy services for specific protocols (such as HTTP), as mentioned in Chapter 14, "Administering Networking." Although it doesn't provide HTTP content itself, a Squid proxy server typically forwards requests from proxy clients to the Internet or other network providing web content. This provides a means of controlling or filtering content that clients can reach from a home, school, or place of business.

webalizer Contains tools for analyzing web server data.

Optional packages in the Web Server group come from the web-server sub-group. Run `yum groupinfo web-server` to display those packages. Some of those packages offer special ways of providing content, such as wikis (`moin`), content management systems (`drupal7`), and blogs (`wordpress`). Others include tools for graphing web statistics (`awstats`) or offer lightweight web server alternatives to Apache (`lighttpd` and `cherokee`).

Installing Apache

Although you only need `httpd` to get started with an Apache web server, if you are just learning about Apache, you should install the manuals (`httpd-manual`) as well. If you are thinking of creating a secure (SSL) site and possibly generating some statistics about your website, you can just install the entire group in Fedora 30:

```
# yum groupinstall "Web Server"
```

Assuming that you have an Internet connection to the Fedora repository (or RHEL repository, if you are using RHEL), all of the mandatory and default packages from that group are installed. You have all of the software that you need to do the procedures and exercises described in this chapter.

Starting Apache

To get the Apache web server going, you want to enable the service to start on every reboot, and you want to start it immediately. In Red Hat Enterprise Linux (up to RHEL 6) and in older Fedora distributions, you could type the following as root:

```
# chkconfig httpd on
# service httpd start
Starting httpd:                [  OK  ]
```

In Fedora 30 and RHEL 8 systems, you enable and start `httpd` using the `systemctl` command:

```
# systemctl enable httpd.service
# systemctl start httpd.service
# systemctl status httpd.service
• httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled;
    vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/httpd.service.d
           └─php-fpm.conf
   Active: active (running) since Mon 2019-09-02 16:16:56 EDT;
    21min ago
     Docs: man:httpd.service(8)
 Main PID: 11773 (/usr/sbin/httpd)
   Status: "Total requests: 14; Idle/Busy workers 100/0;Requests/sec:
    0.0111; Bytes served/s>
    Tasks: 214 (limit: 2294)
   Memory: 24.6M
   CGroup: /system.slice/httpd.service
           ├─11773 /usr/sbin/httpd -DFOREGROUND
           ├─11774 /usr/sbin/httpd -DFOREGROUND
           ├─11775 /usr/sbin/httpd -DFOREGROUND
           ├─11776 /usr/sbin/httpd -DFOREGROUND
           ├─11777 /usr/sbin/httpd -DFOREGROUND
           └─11778 /usr/sbin/httpd -DFOREGROUND
    ...
```

When the `httpd` service starts, five or six `httpd` daemon processes are launched by default (depending on your Linux system) to respond to requests for the web server. You can configure more or fewer `httpd` daemons to be started based on settings in the `httpd.conf` file (described in the section "Understanding the Apache configuration files" later in this chapter).

To change the behavior of the `httpd` daemon, you can edit the `httpd` service by running `systemctl edit httpd`.

Because there are different versions of `httpd` around, check the man page (`man httpd`) to see what options can be passed to the `httpd` daemon. For example, run `systemctl edit httpd` and add an entry as follows:

```
[Service]
Environment=OPTIONS='-e debug'
```

Save the changes (Ctrl+O, Ctrl+X). Adding `-e debug` increases the log level so that the maximum number of Apache messages are sent to log files. Restart the `httpd` service for the changes to take effect. Type the `ps` command to make sure that the options took effect:

```
$ ps -ef | grep httpd
root   14575 1     0 08:49 ? 00:00:01 /usr/sbin/httpd -e debug
-DFOREGROUND
apache 14582 14575 0 08:49 ? 00:00:00 /usr/sbin/httpd -e debug
-DFOREGROUND
```

If you added a debug option (`-e debug`), remember to remove that option by running `systemctl edit httpd` again and removing the entry when you are done debugging Apache, and restart the service. Leaving debugging on will quickly fill up your log files.

Securing Apache

To secure Apache, you need to be aware of standard Linux security features (permissions, ownership, firewalls, and Security Enhanced Linux) as well as security features that are specific to Apache. The following sections describe security features that relate to Apache.

Apache file permissions and ownership

The `httpd` daemon process runs as the user `apache` and group `apache`. By default, HTML content is stored in the `/var/www/html` directory (as determined by the value of `DocumentRoot` in the `httpd.conf` file).

For the `httpd` daemon to be able to access that content, standard Linux permissions apply: If read permission is not on for "other" users, it must be on for the `apache` user or group for the files to be read and served to clients. Likewise, any directory the `httpd` daemon must traverse to get to the content must have execute permission on for the `apache` user, `apache` group, or other user.

Although you cannot log in as the `apache` user (`/sbin/nologin` is the default shell), you can create content as root and change its ownership (`chown` command) or permission (`chmod` command). Often, however, separate user or group accounts are added to create content that is readable by everyone (other) but only writable by that special user or group.

Apache and firewalls

If you have locked down your firewall in Linux, you need to open several ports for clients to be able to talk to Apache through the firewall. Standard web service (HTTP) is accessible over TCP port 80; secure web service (HTTPS) is accessible via TCP port 443. (Port 443 only appears if you have installed the `mod_ssl` package, as described later.)

To verify which ports are being used by the `httpd` server, use the `netstat` command:

```
# netstat -tupln | grep httpd
tcp6   0      0 :::80           :::*                LISTEN      29169/httpd
tcp6   0      0 :::443          :::*                LISTEN      29169/httpd
```

The output shows that the `httpd` daemon (process ID 29169) is listening on all addresses for port 80 (`:::80`) and port 443 (`:::443`). Both ports are associated with the TCP protocol (`tcp6`). To open those ports in Fedora or Red Hat Enterprise Linux, you need to add some firewall rules.

On a current Fedora 30 or RHEL 7 or 8 system, open the Firewall window (type **Firewall** and press Enter from the Activities screen on the GNOME 3 desktop). From there, select Permanent as the configuration. Then, with the public zone selected, click the check boxes next to the http and https service boxes. Those ports immediately become open.

For RHEL 6 or older Fedora releases, add rules to the `/etc/sysconfig/iptables` file (somewhere before a final `DROP` or `REJECT`) such as the following:

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 443 -j ACCEPT
```

Restart `iptables` (`service iptables restart`) for the new rules to take effect.

Apache and SELinux

If *Security Enhanced Linux (SELinux)* is set to enforcing (as it is by default in Fedora and Red Hat Enterprise Linux), SELinux adds another layer of security over your `httpd` service. In essence, SELinux actually sets out to protect the system from being damaged by someone who may have cracked the `httpd` daemon. SELinux does this by creating policies that do the following:

- Deny access to files that are not set to the right file contexts. For `httpd` in SELinux, there are different file contexts for content, configuration files, log files, scripts, and other `httpd`-related files. Any file that is not set to the proper context is not accessible to the `httpd` daemon.
- Prevent insecure features from being used, such as file uploading and clear-text authentication, by setting Booleans for such features to the off position. You can selectively turn on Booleans as they are needed—if they meet your security requirements.
- Keep the `httpd` daemon from accessing nonstandard features, such as a port outside of the default ports the service would expect to use.

A full description of SELinux is contained in Chapter 24, "Enhancing Linux Security with SELinux." However, here are a few specifics you should know about using SELinux with the Apache `httpd` service:

Turn off SELinux You don't have to use SELinux. You can set SELinux to permissive mode if you feel that it is too difficult and unnecessary to create the SELinux policies needed to get your web server to work with SELinux in enforcing mode. You can change the mode to permissive by editing the `/etc/sysconfig/selinux` file so that the `SELINUX` value is set as follows. With this set, the next time you reboot the system, it is in permissive mode. This means that if you break SELinux policies, that event is logged but not prevented (as it would be in enforcing mode).

```
SELINUX=permissive
```

Read the httpd_selinux man page Type **man httpd_selinux** from the shell. This man page shows you the proper file contexts and available Booleans. (If the man page is not there, install it with `yum install selinux-policy-doc`.)

Use standard locations for files When you create new files, those files inherit the file contexts of the directories in which they are stored. Because `/etc/httpd` is set to the right file context for configuration files, `/var/www/html` is right for content files, and so on. Simply copying files to or creating new files in those locations causes the file contexts to be set properly.

Modify SELinux to allow non-standard features You may want to serve web content from the `/mystuff` directory or put configuration files in the `/etc/whatever` directory. Likewise, you may want to allow users of your server to upload files, run scripts, or enable other features that are disabled by SELinux by default. In those cases, you can use SELinux commands to set the file contexts and Booleans that you need to get SELinux working the way you want.

Be sure to read Chapter 24, "Enhancing Linux Security with SELinux," to learn more about SELinux.

Understanding the Apache configuration files

The configuration files for Apache HTTPD are incredibly flexible, meaning that you can configure the server to behave in almost any manner you want. This flexibility comes at the cost of increased complexity in the form of a large number of configuration options (called *directives*). In practice, however, you need to be familiar with only a few directives in most cases.

NOTE

See `http://httpd.apache.org/docs/current/mod/directives.html` for a complete list of directives supported by Apache. If you have `httpd-manual` installed, you can reach descriptions of these directives and other Apache features by opening the manual from the server you have running Apache: `http://localhost/manual/`.

In Fedora and RHEL, the basic Apache server's primary configuration file is in `/etc/httpd/conf/httpd.conf`. Besides this file, any file ending in `.conf` in the `/etc/httpd/conf.d` directory is also used for Apache configuration (based on an `Include` line in the `httpd.conf` file). In Ubuntu, the Apache configuration is stored in text files read by the Apache server, beginning with `/etc/apache2/apache2.conf`. Configuration is read from start to finish, with most directives being processed in the order in which they are read.

Using directives

The scope of many configuration directives can be altered based on context. In other words, some parameters may be set on a global level and then changed for a specific file, directory, or virtual host. Other directives are always global in nature, such as those specifying on which IP addresses the server listens. Still others are valid only when applied to a specific location.

Locations are configured in the form of a start tag containing the location type and a resource location, followed by the configuration options for that location, and finishing with an end tag. This form is often called a *configuration block*, and it looks very similar to HTML code. A special type of configuration block, known as a *location block*, is used to limit the scope of directives to specific files or directories. These blocks take the following form:

```
<locationtag specifier>
(options specific to objects matching the specifier go within this
block)
</locationtag>
```

Different types of location tags exist and are selected based on the type of resource location that is being specified. The specifier included in the start tag is handled based on the type of location tag. The location tags that you generally use and encounter are `Directory`, `Files`, and `Location`, which limit the scope of the directives to specific directories, files, or locations, respectively.

- `Directory` tags are used to specify a path based on the location on the filesystem. For instance, `<Directory />` refers to the root directory on the computer. Directories inherit settings from directories above them, with the most specific `Directory` block overriding less-specific ones, regardless of the order in which they appear in the configuration files.
- `Files` tags are used to specify files by name. `Files` tags can be contained within a `Directory` block to limit them to files under that directory. Settings within a `Files` block override the ones in `Directory` blocks.
- `Location` tags are used to specify the URI used to access a file or directory. This is different from `Directory` in that it relates to the address contained within the request and not to the real location of the file on the drive. `Location` tags are processed last and override the settings in `Directory` and `Files` blocks.

Match versions of these tags—`DirectoryMatch`, `FilesMatch`, and `LocationMatch`—have the same function but can contain regular expressions in the resource specification. `FilesMatch` and `LocationMatch` blocks are processed at the same time as `Files` and `Location`, respectively. `DirectoryMatch` blocks are processed after `Directory` blocks.

Apache can also be configured to process configuration options contained within files with the name specified in the `AccessFileName` directive (which is generally set to `.htaccess`). Directives in access configuration files are applied to all objects under the directory they contain, including subdirectories and their contents. Access configuration files are processed at the same time as `Directory` blocks, using a similar "most specific match" order.

Note

Access control files are useful for allowing users to change specific settings without having access to the server configuration files. The configuration directives permitted within an access configuration file are determined by the `AllowOverride` setting on the directory in which they are contained. Some directives do not make sense at that level and generally result in a "server internal error" message when trying to access the URI. The `AllowOverride` option is covered in detail at `http://httpd.apache.org/docs/mod/core.html#allowoverride`.

Three directives commonly found in location blocks and access control files are `DirectoryIndex`, `Options`, and `ErrorDocument`:

- `DirectoryIndex` tells Apache which file to load when the URI contains a directory but not a filename. This directive doesn't work in `Files` blocks.
- `Options` is used to adjust how Apache handles files within a directory. The `ExecCGI` option tells Apache that files in that directory can be run as CGI scripts, and the `Includes` option tells Apache that server-side includes (SSIs) are permitted. Another common option is the `Indexes` option, which tells Apache to generate a list of files if one of the filenames found in the `DirectoryIndex` setting is missing. An absolute list of options can be specified, or the list of options can be modified by adding `+` or `-` in front of an option name. See `http://httpd.apache.org/docs/mod/core.html#options` for more information.
- `ErrorDocument` directives can be used to specify a file containing messages to send to web clients when a particular error occurs. The location of the file is relative to the `/var/www` directory. The directive must specify an error code and the full URI for the error document. Possible error codes include `403` (access denied), `404` (file not found), and `500` (server internal error). You can find more information about the `ErrorDocument` directive at `http://httpd.apache.org/docs/mod/core.html#errordocument`. As an example, when a client requests a URL from the server that is not found, the following `ErrorDocument` line causes the `404` error code to send the client an error message that is listed in the `/var/www/error/HTTP_NOT_FOUND.html.var` file.

```
ErrorDocument 404 /error/HTTP_NOT_FOUND.html.var
```

Another common use for location blocks and access control files is to limit or expand access to a resource. The `Allow` directive can be used to permit access to matching hosts, and the `Deny` directive can be used to forbid it. Both of these options can occur more than once within a block and are handled based on the `Order` setting. Setting `Order` to `Deny`, `Allow` permits access to any host that is not listed in a `Deny` directive. A setting of `Allow`, `Deny` denies access to any host not allowed in an `Allow` directive.

As with most other options, the most specific Allow or Deny option for a host is used, meaning that you can Deny access to a range and Allow access to subsets of that range. By adding the Satisfy option and some additional parameters, you can add password authentication. For more information on Allow or Deny, Satisfy, or other directives, refer to the Apache Directive Index: http://httpd.apache.org/docs/current/mod/directives.html.

Understanding default settings

The reason you can start using your Apache web server as soon as you install it is that the httpd.conf file includes default settings that tell the server where to find web content, scripts, log files, and other items that the server needs to operate. It also includes settings that tell the server how many server processes to run at a time and how directory contents are displayed.

If you want to host a single website (such as for the example.com domain), you can simply add content to the /var/www/html directory and add the address of your website to a DNS server so that others can browse to it. You can then change directives, such as those described in the previous section, as needed.

To help you understand the settings that come in the default httpd.conf file, I've displayed some of those settings with descriptions below. I have removed comments and rearranged some of the settings for clarity.

The following settings show locations where the httpd server is getting and putting content by default:

```
ServerRoot "/etc/httpd"
Include conf.d/*.conf
ErrorLog logs/error_log
CustomLog "logs/access_log" combined
DocumentRoot "/var/www/html"
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
```

The ServerRoot directive identifies /etc/httpd as the location where configuration files are stored.

At the point in the file where the Include line appears, any files ending in .conf from the /etc/httpd/conf.d directory are included in the httpd.conf file. Configuration files are often associated with Apache modules (and are often included in the software package with a module) or with virtual host blocks (which you might add yourself to virtual host configurations in separate files). See the section "Adding a virtual host to Apache" later in this chapter.

As errors are encountered and content is served, messages about those activities are placed in files indicated by the ErrorLog and CustomLog entries. From the entries shown here, those logs are stored in the /etc/httpd/logs/error_log and /etc/httpd/logs/access_log directories, respectively. Those logs are also hard linked to the /var/log/httpd directory, so you can access the same file from there as well.

The DocumentRoot and ScriptAlias directives determine where content that is served by your httpd server is stored. Traditionally, you would place an index.html file in the DocumentRoot directory (/var/www/html, by default) as the home page and add other content as needed. The ScriptAlias directive tells the httpd daemon that any scripts requested from the cgi-bin directory should be found in the /var/www/cgi-bin directory. For example, a client could access a script located in /var/www/cgi-bin/script.cgi by entering a URL such as http://example.com/cgi-bin/script.cgi.

In addition to file locations, you can find other information in the httpd.conf file. Here are some examples:

```
Listen 80
User apache
Group apache
ServerAdmin root@localhost
DirectoryIndex index.html index.php
AccessFileName .htaccess
```

The Listen 80 directive tells httpd to listen for incoming requests on port 80 (the default port for the HTTP web server protocol). By default, it listens on all network interfaces, although you could restrict it to selected interfaces by IP address (for example, Listen 192.168.0.1:80).

The User and Group directives tell httpd to run as apache for both the user and group. The value of ServerAdmin (root@localhost, by default) is published on some web pages to tell users where to email if they have problems with the server.

The DirectoryIndex lists files that httpd will serve if a directory is requested. For example, if a web browser requested *http://host/whatever/*, httpd would see whether /var/www/html/whatever/index.html existed and serve it if so. If it didn't exist, in this example, httpd would look for index.php. If that file couldn't be found, the contents of the directory would be displayed. An AccessFileName directive can be added to tell httpd to use the contents of the .htaccess file if it exists in a directory to read in settings that apply to access to that directory. For example, the file could be used to require password protection for the directory or to indicate that the contents of the directory should be displayed in certain ways. For this file to work, however, a Directory container (described next) would have to have AllowOverride opened. (By default, the AllowOverride None setting prevents the .htaccess file from being used for any directives.)

The following Directory containers define behavior when the root directory (/), /var/www, and /var/www/html directories are accessed:

```
<Directory/>
    AllowOverride none
    Require all denied
</Directory>
<Directory "/var/www">
    AllowOverride None
```

```
        # Allow open access:
        Require all granted
    </Directory>
    <Directory "/var/www/html">
        Options Indexes FollowSymLinks
        AllowOverride None
        Require all granted
    </Directory>
```

The first `Directory` container (`/`) indicates that if `httpd` tries to access any files in the Linux filesystem, access is denied. The `AllowOverride none` directive prevents `.htaccess` files from overriding settings for that directory. Those settings apply to any subdirectories that are not defined in other `Directory` containers.

Content access is relaxed within the `/var/www` directory. Access is granted to content added under that directory, but overriding settings is not allowed.

The `/var/www/html Directory` container follows symbolic links and does not allow overrides. With `Require all granted` set, `httpd` doesn't prevent any access to the server.

If all of the settings just described work for you, you can begin adding the content that you want to the `/var/www/html` and `/var/www/cgi-bin html` directories. One reason you might not be satisfied with the default setting is that you might want to serve content for multiple domains (such as `example.com`, `example.org`, and `example.net`). To do that, you need to configure virtual hosts. Virtual hosts, which are described in greater detail in the next section, are a convenient (and almost essential) tool for serving different content to clients based on the server address or name to which a request is directed. Most global configuration options are applied to virtual hosts, but they can be overridden by directives within the `VirtualHost` block.

Adding a virtual host to Apache

Apache supports the creation of separate websites within a single server to keep content separate. Individual sites are configured on the same server in what are referred to as *virtual hosts*.

Virtual hosts are really just a way to have the content for multiple domain names available from the same Apache server. Instead of needing to have one physical system to serve content for each domain, you can serve content for multiple domains from the same operating system.

An Apache server that is doing virtual hosting may have multiple domain names that resolve to the IP address of the server. The content that is served to a web client is based on the name used to access the server.

For example, if a client got to the server by requesting the name `www.example.com`, the client would be directed to a virtual host container that had its `ServerName` set to

respond to www.example.com. The container would provide the location of the content and possibly different error logs or Directory directives from the global settings. This way, each virtual host could be managed as if it were on a separate machine.

To use name-based virtual hosting, add as many VirtualHost containers as you like. Here's how to configure a virtual host:

Note

After you enable your first VirtualHost, your default DocumentRoot (/var/www/html) is no longer used if someone accesses the server by IP address or some name that is not set in a VirtualHost container. Instead, the first VirtualHost container is used as the default location for the server.

1. **In Fedora or RHEL, create a file named /etc/httpd/conf.d/example.org.conf using this template:**

```
<VirtualHost *:80>
    ServerAdmin     webmaster@example.org
    ServerName      www.example.org
    ServerAlias     web.example.org
    DocumentRoot    /var/www/html/example.org/
DirectoryIndex  index.php index.html index.htm
</VirtualHost>
```

   This example includes the following settings:

   - The *:80 specification in the VirtualHost block indicates to what address and port this virtual host applies. With multiple IP addresses associated with your Linux system, the * can be replaced by a specific IP address. The port is optional for VirtualHost specifications but should always be used to prevent interference with SSL virtual hosts (which use port 443 by default).
   - The ServerName and ServerAlias lines tell Apache which names this virtual host should be recognized as, so replace them with names appropriate to your site. You can leave out the ServerAlias line if you do not have any alternate names for the server, and you can specify more than one name per ServerAlias line or have multiple ServerAlias lines if you have several alternate names.
   - The DocumentRoot specifies where the web documents (content served for this site) are stored. Although shown as a subdirectory that you create under the default DocumentRoot (/var/www/html), often sites are attached to the home directories of specific users (such as /home/chris/public_html) so that each site can be managed by a different user.

2. **With the host enabled, use `apachectl` to check the configuration, and then do a `graceful` restart:**

```
# apachectl configtest
Syntax OK
# apachectl graceful
```

Provided that you have registered the system with a DNS server, a web browser should be able to access this website using either `www.example.org` or `web.example.org`. If that works, you can start adding other virtual hosts to the system as well.

Another way to extend the use of your website is to allow multiple users to share their own content on your server. You can enable users to add content that they want to share via your web server in a subdirectory of their home directories, as described in the next section.

NOTE

Keeping individual virtual hosts in separate files is a convenient way to manage virtual hosts. However, you should be careful to keep your primary virtual host in a file that will be read before the others because the first virtual host receives requests for site names that don't match any in your configuration. In a commercial web-hosting environment, it is common to create a special default virtual host that contains an error message indicating that no site by that name has been configured.

Allowing users to publish their own web content

In situations where you do not have the ability to set up a virtual host for every user for whom you want to provide web space, you can easily make use of the `mod_userdir` module in Apache. With this module enabled (which it is not by default), the `public_html` directory under every user's home directory is available to the web at `http://`*`servername/~username/`*.

For example, a user named `wtucker` on `www.example.org` stores web content in `/home/wtucker/public_html`. That content would be available from `http://www.example.org/~wtucker`.

Make these changes to the `/etc/httpd/conf/httpd.conf` file to allow users to publish web content from their own home directories. Not all versions of Apache have these blocks in their `httpd.conf` file, so you might have to create them from scratch:

1. **Create a `<IfModule mod_userdir.c>` block.** Change `chris` to any username you want to allow users to create their own `public_html` directory. You can add multiple usernames.

```
<IfModule mod_userdir.c>
     UserDir enabled chris
     UserDir public_html
</IfModule>
```

2. **Create a `<Directory /home/*/public_html>` directive block and change any settings you like**. This is how the block will look:

```
<Directory "/home/*/public_html">
    Options Indexes Includes FollowSymLinks
    Require all granted
</Directory>
```

3. **Have your users create their own `public_html` directories in their own home directories**.

```
$ mkdir $HOME/public_html
```

4. **Set the execute permission (as root user) to allow the `httpd` daemon to access the home directory**:

```
# chmod +x /home /home/*
```

5. **If SELinux is in enforcing mode (which it is by default in Fedora and RHEL), a proper SELinux file context (`httpd_user_content_t`) should already be set on the following directories so that SELinux allows the `httpd` daemon to access the content automatically: `/home/*/www`, `/home/*/web`, and `/home/*/public_html`. If for some reason the context is not set, you can set it as follows:**

```
ttpd_user_content_t to /home/*/
# chcon -R --reference=/var/www/html/ /home/*/public_html
```

6. **Set the SELinux Boolean to allow users to share HTML content from their home directories**:

```
# setsebool -P httpd_enable_homedirs true
```

7. **Restart or reload the `httpd` service**.

   At this point, you should be able to access content placed in a user's `public_html` directory by pointing a web browser to `http://hostname/~user`.

Securing your web traffic with SSL/TLS

Any data that you share from your website using standard HTTP protocol is sent in clear text. This means that anyone who can watch the traffic on a network between your server and your client can view your unprotected data. To secure that information, you can add certificates to your site (so a client can validate who you are) and encrypt your data (so nobody can sniff your network and see your data).

Electronic commerce applications, such as online shopping and banking, should always be encrypted using either the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) specification. TLS is based on version 3.0 of the SSL specifications, so they are very similar in nature. Because of this similarity—and because SSL is older—the SSL acronym is often used to refer to either variety. For web connections, the SSL connection is established first, and then normal HTTP communication is "tunneled" through it.

Note

Because SSL negotiation takes place before any HTTP communication, name-based virtual hosting (which occurs at the HTTP layer) does not work easily with SSL. As a consequence, every SSL virtual host you configure should have a unique IP address. (See the Apache site for more information: `httpd.apache.org/docs/vhosts/name-based.html`.)

While you are establishing a connection between an SSL client and an SSL server, asymmetric (public key) cryptography is used to verify identities and establish the session parameters and the session key. A symmetric encryption algorithm is then used with the negotiated key to encrypt the data that are transmitted during the session. The use of asymmetric encryption during the handshaking phase allows safe communication without the use of a preshared key, and the symmetric encryption is faster and more practical for use on the session data.

For the client to verify the identity of the server, the server must have a previously generated private key as well as a certificate containing the public key and information about the server. This certificate must be verifiable using a public key that is known to the client.

Certificates are generally digitally signed by a third-party *certificate authority (CA)* that has verified the identity of the requester and the validity of the request to have the certificate signed. In most cases, the CA is a company that has made arrangements with the web browser vendor to have its own certificate installed and trusted by default client installations. The CA then charges the server operator for its services.

Commercial certificate authorities vary in price, features, and browser support, but remember that price is not always an indication of quality. Some popular CAs are InstantSSL (`https://www.instantssl.com`), Let's Encrypt (`https://www.letsencrypt.org`), and DigiCert (`https://www.digicert.com`).

You also have the option of creating self-signed certificates, although these should be used only for testing or when a very small number of people will be accessing your server and you do not plan to have certificates on multiple machines. Directions for generating a self-signed certificate are included in the section "Generating an SSL key and self-signed certificate" later in this chapter.

The last option is to run your own certificate authority. This is probably practical only if you have a small number of expected users and the means to distribute your CA certificate to them (including assisting them with installing it in their browsers). The process for creating a CA is too elaborate to cover in this book, but it is a worthwhile alternative to generating self-signed certificates.

The following sections describe how HTTPS communications are configured by default in Fedora and RHEL when you install the `mod_ssl` package. After that, I describe how to configure SSL communications better by generating your own SSL keys and certificates to use with the web server (running on a Fedora or RHEL system) configured in this chapter.

Understanding how SSL is configured

If you have installed the `mod_ssl` package in Fedora or RHEL (which is done by default if you installed the Basic Web Server group), a self-signed certificate and private key are created when the package is installed. This allows you to use HTTPS protocol immediately to communicate with the web server.

Although the default configuration of `mod_ssl` allows you to have encrypted communications between your web server and clients, because the certificate is self-signed, a client accessing your site is warned that the certificate is untrusted. To begin exploring the SSL configuration for your Apache web server, make sure that the `mod_ssl` package is installed on the server running your Apache `(httpd)` service:

```
# yum install mod_ssl
```

The `mod_ssl` package includes the module needed to implement SSLon your web server (`mod_ssl.so`) and a configuration file for your SSL hosts: `/etc/httpd/conf.d/ssl.conf`. There are many comments in this file to help you understand what to change. Those lines that are not commented out define some initial settings and a default virtual host. Here are some of those lines:

```
Listen 443 https
...
<VirtualHost _default_:443>
ErrorLog logs/ssl_error_log
TransferLog logs/ssl_access_log
LogLevel warn
SSLEngine on
...
SSLCertificateFile /etc/pki/tls/certs/localhost.crt
SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
...
</VirtualHost>
```

The SSL service is set to listen on standard SSL port 443 on all the system's network interfaces.

A `VirtualHost` block is created that causes error messages and access messages to be logged to log files that are separate from the standard logs used by the server (`ssl_error_log` and `ssl_access_log` in the `/var/log/httpd/` directory). The level of log messages is set to `warn` and the SSLEngine is turned on.

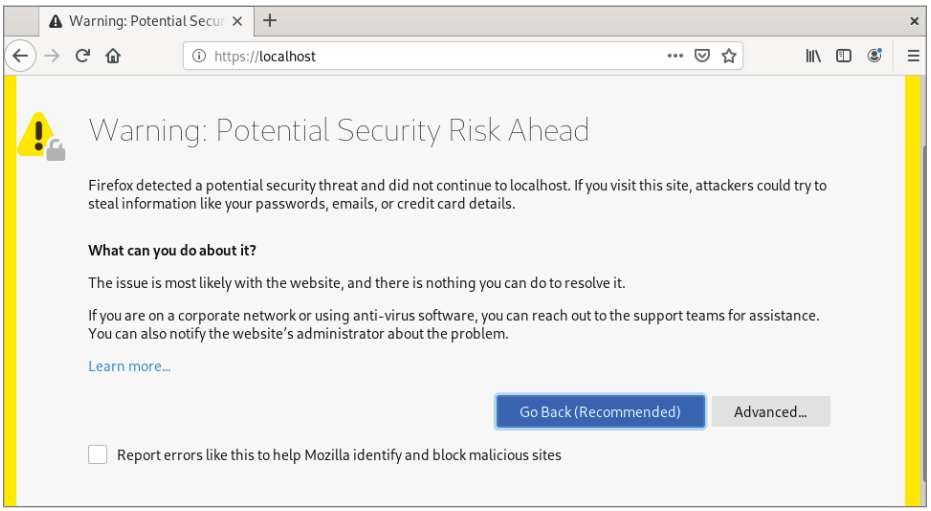
In the preceding sample code, two entries associated with SSL Certificates in the `VirtualHost` block identify the key and certificate information. As mentioned previously, a key is generated when `mod_ssl` is installed and placed in the file `/etc/pki/tls/private/localhost.key`. A self-signed certificate, `/etc/pki/tls/certs/localhost.crt`, is created using that key. When you create your own key and certificate later, you need to replace the values of `SSLCertificateFile` and `SSLCertificateKeyFile` in this file.

After installing the `mod_ssl` package and reloading the configuration file, you can test that the default certificate is working by following these steps:

1. **Open a connection to the website from a web browser, using the `HTTPS` protocol.** For example, if you are running Firefox on the system where the web server is running, type **`https://localhost`** into the location box and press Enter. Figure 17.2 shows an example of the page that appears.

FIGURE 17.2

Accessing an SSL website with a default certificate



2. This page warns you that there is no way of verifying the authenticity of this site. That is because there is no way to know who created the certificate that you are accepting.
3. **Because you are accessing the site via a browser on the local host, click Advanced and then View to see the certificate that was generated.** It includes your hostname, information on when the certificate was issued and when it expires, and lots of other organization information.
4. **Select Accept the Risk and Continue to allow connections to this site.**
5. **Close that window, and then select Confirm Security Exception to accept the connection.** You should now see your default web page using HTTPS protocol. From now on, your browser will accept HTTPS connections to the web server using that certificate and encrypt all communications between the server and browser.

Because you don't want your website to scare off users, the best thing to do is to get a valid certificate to use with your site. The next best thing to do is to create a self-signed certificate that at least includes better information about your site and organization. The following section describes how to do that.

Generating an SSL key and self-signed certificate

To begin setting up SSL, use the `openssl` command, which is part of the `openssl` package, to generate your public and private key. After that, you can generate your own self-signed certificate to test the site or to use internally.

1. **If the `openssl` package is not already installed, install it as follows:**

   ```
   # yum install openssl
   ```

2. **Generate a 2048-bit RSA private key and save it to a file:**

   ```
   # cd /etc/pki/tls/private
   # openssl genrsa -out server.key 2048
   # chmod 600 server.key
   ```


Note

You can use a filename other than `server.key` and should do so if you plan to have more than one SSL host on your machine (which requires more than one IP address). Just make sure that you specify the correct filename in the Apache configuration later.

Or, in higher-security environments, encrypting the key by adding the `-des3` argument after the `genrsa` argument on the `openssl` command line is a good idea. When prompted for a passphrase, press Enter:

```
# openssl genrsa -des3 -out server.key 1024
```

3. **If you don't plan to have your certificate signed, or if you want to test your configuration, generate a self-signed certificate and save it in a file named `server.crt` in the `/etc/pki/tls/certs` directory:**

   ```
   # cd /etc/pki/tls/certs
   # openssl req -new -x509 -nodes -sha1 -days 365 \
      -key /etc/pki/tls/private/server.key \
      -out server.crt
   Country Name (2 letter code) [AU]: US
   State or Province Name (full name) [Some-State]: NJ
   Locality Name (eg, city) [Default City]: Princeton
   Organization Name (eg, company) [Default Company Ltd
   Ltd]:TEST USE ONLY
   Organizational Unit Name (eg, section) []:TEST USE ONLY
   Common Name (eg, YOUR name) []:secure.example.org
   Email Address []:dom@example.org
   ```

4. **Edit the `/etc/httpd/conf.d/ssl.conf` file to change the key and certificate locations to use the ones that you just created.** For example:

   ```
   SSLCertificateFile /etc/pki/tls/certs/server.crt
   SSLCertificateKeyFile /etc/pki/tls/private/server.key
   ```

5. **Restart or reload the `httpd` server.**
6. **Open `https://localhost` from a local browser again, repeat the procedure to review, and accept the new certificate.**

For internal use or testing, a self-signed certificate might work for you. However, for public websites, you should use a certificate that is validated by a certificate authority (CA). The procedure for doing that is covered next.

Generating a certificate signing request

If you plan to have your certificate signed by a CA (including one that you run yourself), you can use your private key to generate a certificate signing request (CSR):

1. **Create a directory for storing your CSR.**

   ```
   # mkdir /etc/pki/tls/ssl.csr
   # cd /etc/pki/tls/ssl.csr/
   ```

2. **Use the `openssl` command to generate the CSR.** The result is a CSR file in the current directory named `server.csr`. When you enter the information, the Common Name entry should match the name that clients will use to access your server. Be sure to get the other details right so that it can be validated by a third-party CA. Also, if you had entered a passphrase for your key, you are prompted to enter it here to use the key.

   ```
   # openssl req -new -key ../private/server.key -out server.csr

   Country Name (2 letter code) [AU]:US
   State or Province Name (full name) [Some-State]:Washington
   Locality Name (eg, city) []:Bellingham
   Organization Name (eg, company) [Internet Widgits Pty
   Ltd]:Example Company, LTD.
   Organizational Unit Name (eg, section) []:Network
    Operations
   Common Name (eg, YOUR name) []:secure.example.org
   Email Address []:dom@example.org

   Please enter the following 'extra' attributes
   to be sent with your certificate request
   A challenge password []:
   An optional company name []:
   ```

3. **Visit the website of the certificate signing authority that you choose and request a signed certificate.** At some point, the CA site will probably ask you to

copy and paste the contents of your CSR (`server.csr` file in this example) into a form needed to make the request.

4. **When the CA sends you the certificate (probably via email), save it in the `/etc/pki/tls/certs/` directory using a name based on the site you are hosting — for example, `example.org.crt`.**
5. **Change the value of `SSLCertificateFile` in the `/etc/httpd/conf.d/ssl.conf` file to point to your new CRT file.** Or, if you have multiple SSL hosts, you might want to create a separate entry (possibly in a separate `.conf` file) that looks like the following:

```
Listen 192.168.0.56:443
<VirtualHost *:443>
    ServerName      secure.example.org
    ServerAlias     web.example.org
    DocumentRoot    /home/username/public_html/
    DirectoryIndex  index.php index.html index.htm
    SSLEngine       On
    SSLCertificateKeyFile /etc/pki/tls/private/server.key
    SSLCertificateFile /etc/pki/tls/certs/example.org.crt
</VirtualHost>
```

The IP address shown in the `Listen` directive should be replaced by the public IP address representing the SSL host you are serving. Remember that each SSL host should have its own IP address.

Troubleshooting Your Web Server

In any complex environment, you occasionally run into problems. The following sections include tips for isolating and resolving the most common errors that you may encounter.

Checking for configuration errors

You may occasionally run into configuration errors or script problems that prevent Apache from starting or that prevent specific files from being accessible. Most of these problems can be isolated and resolved using two Apache-provided tools: the `apachectl` program and the system error log.

When encountering a problem, first use the `apachectl` program with the `configtest` parameter to test the configuration. In fact, it's a good idea to develop the habit of running this every time you make a configuration change:

```
# apachectl configtest
Syntax OK
# apachectl graceful
/usr/sbin/apachectl graceful: httpd gracefully restarted
```

In the event of a syntax error, `apachectl` indicates where the error occurs and also does its best to give a hint about the nature of the problem. You can then use the `graceful` restart option (`apachectl graceful`) to instruct Apache to reload its configuration without disconnecting any active clients.

NOTE
The `graceful` restart option in `apachectl` automatically tests the configuration before sending the reload signal to `apache`, but getting in the habit of running the manual configuration test after making any configuration changes is still a good idea.

Some configuration problems pass the syntax tests performed by `apachectl` but cause the HTTP daemon to exit immediately after reloading its configuration. If this happens, use the `tail` command to check Apache's error log for useful information. On Fedora and RHEL systems, the error log is in `/var/log/httpd/error.log`. On other systems, you can find the location by looking for the `ErrorLog` directive in your Apache configuration.

You might encounter an error message that looks something like this:

```
[crit] (98)Address already in use: make_sock: could not bind to port
80
```

This error often indicates that something else is bound to port 80, that another Apache process is already running (`apachectl` usually catches this), or that you have told Apache to bind the same IP address and port combination in more than one place.

You can use the `netstat` command to view the list of programs (including Apache) with TCP ports in the `LISTEN` state:

```
# netstat -nltp
Active Internet connections (only servers)
Proto  Local Address  Foreign Address  State   PID/Program name
tcp6   :::80          :::*             LISTEN  2105/httpd
```

The output from `netstat` (which was shortened to fit here) indicates that an instance of the `httpd` process with a process ID of `2105` is listening (as indicated by the `LISTEN` state) for connections to any local IP address (indicated by `:::80`) on port 80 (the standard HTTP port). If a different program is listening to port 80, it is shown there. You can use the `kill` command to terminate the process, but if it is something other than `httpd`, you should also find out why it is running.

If you don't see any other processes listening on port 80, it could be that you have accidentally told Apache to listen on the same IP address and port combination in more than one place. Three configuration directives can be used for this: `BindAddress`, `Port`, and `Listen`:

- `BindAddress` enables you to specify a single IP address on which to listen, or you can specify all IP addresses using the `*` wildcard. You should never have more than one `BindAddress` statement in your configuration file.

- `Port` specifies on which TCP port to listen, but it does not enable you to specify the IP address. `Port` is generally not used more than once in the configuration.
- `Listen` enables you to specify both an IP address and a port to bind to. The IP address can be in the form of a wildcard, and you can have multiple `Listen` statements in your configuration file.

To avoid confusion, it is generally a good idea to use only one of these directive types. Of the three, `Listen` is the most flexible, so it is probably the one you want to use the most. A common error when using `Listen` is to specify a port on all IP addresses (`*:80`) as well as that same port on a specific IP address (`1.2.3.4:80`), which results in the error from `make_sock`.

Configuration errors relating to SSL commonly result in Apache starting improperly. Make sure that all key and certificate files exist and that they are in the proper format (use `openssl` to examine them).

For other error messages, try doing a web search to see whether somebody else has encountered the problem. In most cases, you can find a solution within the first few matches.

If you aren't getting enough information in the `ErrorLog`, you can configure it to log more information using the `LogLevel` directive. The options available for this directive, in increasing order of verbosity, are `emerg`, `alert`, `crit`, `error`, `warn`, `notice`, `info`, and `debug`. Select only one of these.

Any message that is at least as important as the `LogLevel` that you select are stored in the `ErrorLog`. On a typical server, `LogLevel` is set to `warn`. You should not set it to any value lower than `crit`, and you should avoid leaving it set to `debug` because that can slow down the server and result in a very large `ErrorLog`.

As a last resort, you can also try running `httpd -X` manually to check for crashes or other error messages. The `-X` runs `httpd` so that it displays debug and higher messages on the screen.

Accessing forbidden and server internal errors

The two common types of errors that you may encounter when attempting to view specific pages on your server are permission errors and server internal errors. Both types of errors can usually be isolated using the information in the error log. After making any of the changes described in the following list to attempt to solve one of these problems, try the request again and check the error log to see whether the message has changed (for example, to show that the operation completed successfully).

Note

"File not found" errors can be checked in the same way as "access forbidden" and "server internal errors." You may sometimes find that Apache is not looking where you think it is for a specific file. Generally, the entire path to the file shows up in the error log. Make sure that you are accessing the correct virtual host, and check for any `Alias` settings that might be directing your location to a place you don't expect.

File permissions A "File permissions prevent access" error indicates that the `apache` process is running as a user that is unable to open the requested file. By default, `httpd` is run by the `Apache` user and group. Make sure that the account has execute permissions on the directory, and every directory above it, as well as read permissions on the files themselves. Read permissions on a directory are also necessary if you want Apache to generate an index of files. See the manual page for `chmod` for more information about how to view and change permissions.

NOTE
Read permissions are not necessary for compiled binaries, such as those written in C or C++, but they can be safely added unless a need exists to keep the contents of the program secret.

Access denied A "Client denied by server configuration" error indicates that Apache was configured to deny access to the object. Check the configuration files for `Location` and `Directory` sections that might affect the file that you are trying to access. Remember that settings applied to a path are also applied to any paths below it. You can override these by changing the permissions only for the more specific path to which you want to allow access.

Index not found The "Directory index forbidden by rule" error indicates that Apache could not find an index file with a name specified in the `DirectoryIndex` directive and was configured not to create an index containing a list of files in a directory. Make sure that your index page, if you have one, has one of the names specified in the relevant `DirectoryIndex` directive, or add an `Options Indexes` line to the appropriate `Directory` or `Location` section for that object.

Script crashed "Premature end of script headers" errors can indicate that a script is crashing before it finishes. On occasion, the errors that caused this also show up in the error log. When using `suexec` or `suPHP`, this error may also be caused by a file ownership or permissions error. These errors appear in log files in the `/var/log/httpd` directory.

SELinux errors If file permissions are open but messages denying permission appear in log files, SELinux could be causing the problem. Set SELinux to permissive mode temporarily (`setenforce 0`) and try to access the file again. If the file is now accessible, set SELinux to enforcing mode again (`setenforce 1`) and check file contexts and Booleans. File contexts must be correct for `httpd` to be able to access a file. A Boolean might prevent a file being served from a remotely mounted directory or prevent a page from sending an email or uploading a file. Type **`man httpd_selinux`** for details about SELinux configuration settings associated with the `httpd` services. (Install the `selinux-policy-devel` package to have that man page added to your system.)

Summary

The open source Apache project is the world's most popular web server. Although Apache offers tremendous flexibility, security, and complexity, a basic Apache web server can be configured in just a few minutes in Fedora, RHEL, and most other Linux distributions.

The chapter described the steps for installing, configuring, securing, and troubleshooting a basic Apache web server. You learned how to configure virtual hosting and secure SSL hosts. You also learned how to configure Apache to allow any user account on the system to publish content from their own `public_html` directory.

Continuing on the topic of server configuration, in Chapter 18, "Configuring an FTP Server," you will learn how to set up an FTP server in Linux. The examples illustrate how to configure an FTP server using the `vsftpd` package.

Exercises

The exercises in this section cover topics related to installing and configuring an Apache web server. As usual, I recommend that you use a spare Fedora or Red Hat Enterprise Linux system to do the exercises. Don't do these exercises on a production machine because these exercises modify the Apache configuration files and service, and they could damage services that you have currently configured. Try to use a virtual machine or find a computer where it will do no harm to interrupt services on the system.

These exercises assume that you are starting with a Fedora or RHEL installation on which the Apache server (`httpd` package) is not yet installed.

If you are stuck, solutions to the tasks are shown in Appendix B. These show you one approach to each task, although Linux may offer multiple ways to complete a task.

1. From a Fedora system, install all of the packages associated with the Basic Web Server group.
2. Create a file called `index.html` in the directory assigned to `DocumentRoot` in the main Apache configuration file. The file should have the words "My Own Web Server" inside.
3. Start the Apache web server and set it to start up automatically at boot time. Check that it is available from a web browser on your local host. (You should see the words "My Own Web Server" displayed if it is working properly.)
4. Use the `netstat` command to see on which ports the `httpd` server is listening.
5. Try to connect to your Apache web server from a web browser that is outside of the local system. If it fails, correct any problems that you encounter by investigating the firewall, SELinux, and other security features.

6. Using the `openssl` or similar command, create your own private RSA key and self-signed SSL certificate.
7. Configure your Apache web server to use your key and self-signed certificate to serve secure (HTTPS) content.
8. Use a web browser to create an HTTPS connection to your web server and view the contents of the certificate that you created.
9. Create a file named `/etc/httpd/conf.d/example.org.conf`, which turns on name-based virtual hosting and creates a virtual host that does these things:
   - Listens on port 80 on all interfaces
   - Has a server administrator of `joe@example.org`
   - Has a server name of `joe.example.org`
   - Has a `DocumentRoot` of `/var/www/html/example.org`
   - Has a `DirectoryIndex` that includes at least `index.html`

   Create an `index.html` file in `DocumentRoot` that contains the words "Welcome to the House of Joe" inside.
10. Add the text `joe.example.org` to the end of the localhost entry in your `/etc/hosts` file on the machine that is running the web server. Then type `http://joe.example.org` into the location box of your web browser. You should see "Welcome to the House of Joe" when the page is displayed.